CodeWarrior Linker Command File (LCF) for Kinetis

1 Introduction

This document provides the steps to create LCF from scratch and explains common as well as unique application requirements handled in LCF using examples.

For detailed information on the Kinetis Linker Command File (LCF) refer to the CodeWarrior Development Studio for Kinetis Build Tools Reference Manual. You can find this document in `{MCU10.xinstallation path}\MCU\Help\PDF\MCU_Kinetis_Compiler.pdf`

2 Preliminary Background

The Linker Command File along with other compiler directives, places pieces of code and data into ROM and RAM. The compiler ELF file output defines default linker input sections, normally named `.text`, `.data`, `.bss`, `.rodata` and the linker, via the LCF, is used to combine input sections into output sections and loadable memory segments.

Contents

The user may define customized linker input sections in their source code using compiler pragma directives. The LCF can be instructed to consume these customized sections. LCF consists of three kinds of segments, which must be in this order:

- A memory segment, which begins with the `MEMORY{}` directive,
- An optional closure segment, which begins with the `FORCE_ACTIVE{}`, `KEEP_SECTION{}`, or `REF_INCLUDE{}` directives, and
- A sections segment, which begins with the `SECTIONS{}` directive.

3 Creating an LCF for a Sample ROM Project

First add the memory area for interrupts, code, data and some flash data.

NOTE The memory regions and LCF explained are with respect to PK60N512.

Listing 1. Memory Segment

```
MEMORY
{
    m_interrupts  (RX) : ORIGIN = 0x00000000, LENGTH = 0x000001E0
    m_text        (RX) : ORIGIN = 0x00000800, LENGTH = 0x0007F800
    m_data        (RW) : ORIGIN = 0x1FFF0000, LENGTH = 0x00020000
    m_cfmprotrom  (RX) : ORIGIN = 0x00000400, LENGTH = 0x00000010
}
```

You can notice that the LCF defines four Microcontroller memory segments. Each default segment allocates the following information:

- `m_interrupts` — Exception handler vectors.
- `Cfmprotrom` — Flash backdoor key.
- `m_text` — Application's instructions/code.
- `m_data` — initialized and uninitialized data

NOTE For more information, refer **LCF Structure > Memory Segment and Commands, Directives, and keywords > MEMORY** directive topics in the ELF Linker and Command Language chapter of Microcontrollers V10.x Kinetis Build Tools Reference Manual.

Place the sections to the above memory areas in LCF in the SECTIONS { } block. A section can be named as you want.

The sections segment of a LCF created by a stationary project contains seven default sections.

- `.app_text` — Contains all the code and constants of the application. This section is located in segment m_text.
- `.interrupts` — Is placed in segment m_interrupts and contains the exceptions vector table.
- `.cfmprotect` — Contains the flash backdoor key. It is contained in cfmprotrom.

- `.app_data` — Contains all the data needed by the program while its execution. This section is located in segment m_data.
- `.bss` — Contains non-initialized data and is located in segment m_data.
- `.romp` — When a project starts running, all the variables are loaded into RAM in order to be used. This data must be located in ROM and then it must be copied to RAM in runtime. This section contains a structure that is used in startup routine to copy data in section .data from ROM to RAM.

To understand the content of sections following are the LCF sections that are used and modified in order to relocate code and data.

Listing 2. Section .interrupts

```
.interrupts :
{
    __vector_table = .;
    * (.vectortable)
    . = ALIGN (0x4);
} > m_interrupts
```

The hardware initialization routines, application code and constants are grouped together and placed in Flash.

Listing 3. Section .app_text

```
    .app_text:
{
       ALIGNALL(4);
       * (.init)
       * (.text)
       .= ALIGN(0x8) ;
       * (.rodata)
       .= ALIGN(0x4) ;
    ___ROM_AT = .;
} > m_text
```

`.app_text` is the name of the output section in the executable file. It can be any name as you want. .text, .init and .rodata are the input section names present in each of the object files linked. Asterisk(*) refers to all the .text sections of the object files. A filename can also be specified instead of asterisk to link only the .text section of the specified filename to this section.

After all the code is placed into memory and the memory gets aligned, the read only data is placed next to the code with the instruction `*(.rodata)`.

The label `___ROM_AT` point to the first empty address after `*(.text)` and `*(.rodata)` are placed.

Finally, > m_text means that all the content of this section is going to be placed in memory segment m_text.

The initialized data and C++ exception tables are placed in RAM as shown below.

Listing 4. Section .app_data

```
.app_data: AT(___ROM_AT)
{
       * (.sdata)
```

```
        * (.data)
        .= ALIGN(0x4) ;
        *(.ARM.extab)
        .= ALIGN(0x4) ;
        __exception_table_start__ = .;
        EXCEPTION
        __exception_table_end__ = .;
        .= ALIGN(0x4) ;
        __sinit__ = .;
        STATICINIT
        .= ALIGN(0x8) ;
} > m_data
```

The first line contains the instruction AT with the label `___ROM_AT` as parameter. This means that the content of section `.app_data` is going to be resident in the flash address pointed by label `___ROM_AT`. Then, the startup code performs a routine to copy this data from Flash to RAM using the information provided in section `.romp`. This routine can be found in the `startup.c` file.

The uninitialized data placed in RAM following the `m_data`.

Listing 5. Section .bss

```
.bss :
{
        .= ALIGN(0x4) ;
        __START_BSS = .;
        * (.bss)
        __END_BSS = .;
        .= ALIGN(0x8) ;
} >> m_data
```

Listing 6. Section .app_data

```
_romp_at = ___ROM_AT + SIZEOF(.app_data);
.romp : AT(_romp_at)
{
    __S_romp = _romp_at;  /* structure address used to copy ROM to RAM, if 0 there is no ROM
to RAM copy */
    WRITEW(___ROM_AT); /* start address of .app_data in ROM */
    WRITEW(ADDR(.app_data)); /*Address of .app_data in RAM, where the data need to be copied */
    WRITEW(SIZEOF(.app_data));
    WRITEW(0);
    WRITEW(0);
    WRITEW(0);
}
```

NOTE The section details are provided in `MCU_Kinetis_Compiler.pdf`.

The sections are allocated to segments in the order given in SECTIONS block of lcf file.

Variables are added in LCF and these can be used in application as well as internally in linker tool for computation.

Listing 7. Adding Variables

```
__SP_INIT = . + 0x00008000;
__heap_addr = __SP_INIT;
__heap_size = 0x00008000;
```

Let us take a simple example to see how the allocation of variables to the respective sections takes place.

Listing 8. C Source file:

```
 const char rodata_array[40]="CodeWarior";
 long bss_i;
 long data_i = 10;

int main(void) {
  return data_i + bss_i + rodata_array[5];
}
```

NOTE Above is a hypothetical example built to provide clarity on variables and their allocation to sections.

The objects are allocated to the sections as given in Table:

Table 1. Object and its allocation

Variable	Input Section	Address	Output Section
data_i	.data	1FFF0000	.app_data
bss_i	.bss	1FFF0008	.bss
rodata_array	.rodata	00000A88	.app_text

4 Relocating Code in ROM

To place data and code in a specific memory location there are two general steps that must be performed.

1. Use pragma compiler directives to tell the compiler which part of the code is going to be relocated.
2. Tell the linker where is going to be placed this code within the memory map using LCF definitions.

4.1 Relocating Function in ROM

To put code in a specific memory section it is needed first to create the section using the define_section pragma directive. In Listing 9 a new section called mySectionInROM is created.

All the content in this section is going to be referenced in the LCF with the name `.romsymbols`.

After defining a new section you can place code in this section by using the `__declspec()` directive.

In Listing `9, __declspec()` directive is used to tell the compiler that function `funcInROM()` is going to be placed in section `mySectionInROM`.

Create a stationary project for PK60N512 and add the following code to your main.c file before the main() function and have a call to this function.

Listing 9. Code to add in the main.c

```
#pragma define_section mySectionInROM ".romsymbols" far_abs RX
__declspec(section "mySectionInROM") void funcInROM(int flag); //Fcn Prototype
 void funcInROM(int flag){
if (flag > 0)
{
printf("Option 1 selected \n\r");
printf("Executing funcInROM() \n\r");
printf("This function is executed from section myROM \n\r");
}
}
```

4.2 Placing Code in ROM

You have just edited a source file to tell the compiler which code will be relocated. Next, the LCF needs to be edited to tell the linker the memory addresses where these sections are going to be allocated.

First you need to define a new Microcontroller memory segment where new sections will be allocated.

You can have just one memory segment for all the new sections or one segment for each section.

4.2.1 Create New ROM Segment

Below you can find the memory segment of a LCF. Notice that the segment code has been edited and its length has been reduced by 0x10000 from its original size. This memory space is taken to create the new segment. In the following listing the new segment is called myrom, it will be located next to segment code and its length is going to be 0x10000. You can calculate the address where segment code ends by adding its length plus the origin address.

Edit your LCF as shown in the following listing. Ensure you edit PK60N512_flash.lcf.

Listing 10. Memory Segment of LCF

```
MEMORY {
    m_interrupts  (RX) : ORIGIN = 0x00000000, LENGTH = 0x000001E0
    m_text        (RX) : ORIGIN = 0x00000800, LENGTH = 0x0006F800
    myrom (RX) : ORIGIN = 0x00070000, LENGTH = 0x00010000
    m_data        (RW) : ORIGIN = 0x1FFF0000, LENGTH = 0x00020000
    m_cfmprotrom  (RX) : ORIGIN = 0x00000400, LENGTH = 0x00000010
}
```

4.2.2 Create New ROM Section

The next step is to add the content of the new section into the Microcontroller memory segment you have reserved. This is done in the sections segment of the LCF.

The code below creates a new section called .rom_symbols, then the label `__ROM_SYMBOLS` points to the address where the section begins. Then `*(.romsymbols)` instruction is used to tell the linker that all the code referenced with this word is going to be placed in section .rom_symbols.

Finally you close the section telling the linker that this content is going to be located in segment myrom.

Add the code as in Listing 11 just after section `.app_text` and before section `.app_data` in your `PK60N512_flash.lcf`.

Listing 11. Code to Add after Section .app_text

```
.rom_symbols :
{
__ROM_SYMBOLS = . ; #start address of the new symbol area
. = ALIGN (0x4);
*(.romsymbols) #actual data matching pragma directives.
. = ALIGN (0x4);
} > myrom
```

Inside your project folder you will find a subfolder called PK60N512_Flash. In this folder you will find your .xMAP file. If you compile at this point you will find in the xMAP file that section .rom_symbols was created in address 0x70000 as shown in the following listing.

Listing 12. .rom_symbols Created in Address 0x70000

```
# .rom_symbols
#>00070000          __ROM_SYMBOLS (linker command file)
  00070000 00000000 .romsymbols $t                   (main.obj)
  00070000 00000038 .romsymbols funcInROM            (main.obj)
  0007002C 00000000 .romsymbols $d(main.obj)
```

NOTE The label is pointing to the start address and funcInROM has been linked to `.rom_symbols`.

5 Relocating Code and Data in Internal RAM

Since it is not possible to write a variable in ROM, data must be relocated in RAM. Code can be also relocated in RAM. Another reason to relocate code in RAM is that it is twice as fast as in Flash.

In Relocating Code in ROM you have seen how to put one single function in a new section using __declspec() directive. As it was only one function it was not a problem. In big projects it may be complicated to use this directive with each of the functions you want to relocate.

5.1 Relocating Code and Data in Internal RAM

Another way to tell the compiler to relocate code and also data is to create a new section using define_section pragma directive and writing the code inside `#pragma section <Section Name> begin` and `#pragma section <Section Name> end directives` as shown in the following listing.

Listing 13. Using Pragma Directives to Define a Section

```
#pragma define_section mySectionInRAM ".myCodeInRAM" far_abs RX
#pragma section mySectionInRAM begin
struct {
unsigned char data0;
unsigned char data1;
unsigned char data2;
unsigned char data3;
unsigned char data4;
unsigned char data5;
unsigned char data6;
unsigned char data7;
} CTMData = { 0x82, 0x65, 0x77, 0x32, 0x84, 0x69, 0x83, 0x84 };
void funcInROM(int flag);
 void funcInROM(int flag){
if (flag > 0)
{
printf("Option 1 selected \n\r");
printf("Executing funcInROM() = %c \n\r", CTMData.data1);
printf("This function is executed from section myROM \n\r");
}
}
#pragma section mySectionInRAM end
```

5.2 Placing Code and Data in RAM

Placing code and data into RAM is more complicated. As the content in RAM cannot be saved when turning power off, you first need to save the code and data in flash and then make a copy to RAM in runtime.

Next are described the steps to relocate code and data in a new RAM segment.

5.2.1 Create New RAM Segment

As it was made for the new ROM segment, a piece of the m_data memory segment is taken to create a new memory segment called myram.

Edit your LCF as shown in Listing 14.

Listing 14. Edit LCF

```
MEMORY {
m_interrupts  (RX) : ORIGIN = 0x00000000, LENGTH = 0x000001E0
m_text        (RX) : ORIGIN = 0x00000800, LENGTH = 0x0006F800
m_data        (RW) : ORIGIN = 0x1FFF0000, LENGTH = 0x00010000
myram (RX) : ORIGIN = 0x20000000, LENGTH = 0x00010000
m_cfmprotrom  (RX) : ORIGIN = 0x00000400, LENGTH = 0x00000010
}
```

5.2.2 Create New RAM Section

The memory segment specifies the intended location in RAM. The sections segment specifies the resident location in ROM, via its AT (address) parameter. The code below shows a new section called `.my_ram` which is going to be linked in segment `myram` but is going to be resident in the Flash memory address calculated by label `___CodeStart`. This label is intended to find the first address available in flash.

In the listing Section .app_data the linker places in the segment code all the code and then the read only data. After this it sets a label called `__ROM_AT`. Section `.app_data` is allocated in the address pointed by this label. You can place the new code after section .app_data, so you calculate the address where section `.app_data` ends by using the instruction

`___ROM_AT + SIZEOF(.app_data).`

Add the code in the following listing to the LCF. You can put this code just after section `.app_data`.

Listing 15. Add this Code to LCF after .app_data

```
___CodeStart = ___ROM_AT + SIZEOF(.app_data);
.my_ram : AT(___CodeStart)
{
. = ALIGN (0x4);
___myRAMStart = .;
*(.myCodeInRAM)
___myRAMEnd = .;
. = ALIGN (0x4);
} > myram
___CodeSize = ___myRAMEnd - ___myRAMStart
```

You need to know the size of the code to indicate the amount of bytes that are going to be copied from

Flash to RAM in runtime. To do this three labels are defined in the code above.

- `___myRAMStart` — is set in the point just before the new code is placed
- `___myRAMEnd` — is set after the code indicating the address where it ends
- `___CodeSize` — Subtraction of both the addresses gives us the size of the code and this value is assigned to the label ___CodeSize.

In order to let the compiler know the value of these labels the following defines must be declared in the source files. Copy the code in Listing 16 in main.c file before all the functions.

Listing 16. Code to Copy in main.c

```
extern unsigned long __CodeStart[];
#define CodeStart (unsigned long)__CodeStart
extern unsigned long __CodeSize[];
#define CodeSize (unsigned long)__CodeSize
extern unsigned long __myRAMStart[];
#define StartAddr (unsigned long)__myRAMStart
```

5.2.3 Final Adjustments

Section `.romp` provides to the startup routine the information needed to copy the content of section `.app_data` from Flash to RAM. Notice that this section is placed by default after section `.app_data`. This is indicated by the instruction AT(`_romp_at`).

Label `_romp_at` points to the address where section .app_data ends, but now you have placed the section `.my_ram` in that memory space. As you can notice labels `___CodeStart` and `_romp_at` point to the same address. So you need to move section .romp after section `.my_ram` by adding the space occupied by this section. This is done with the instruction `SIZEOF(.my_ram)`.

Listing 17. Modifying the LCF

```
_romp_at = ___ROM_AT + SIZEOF(.app_data)+SIZEOF(.my_ram);
.romp : AT(_romp_at)
{
    __S_romp = _romp_at;
    WRITEW(___ROM_AT);
    WRITEW(ADDR(.app_data));
    WRITEW(SIZEOF(.app_data));
    WRITEW(0);
    WRITEW(0);
    WRITEW(0);
}
```

NOTE The .romp structure is actually an array of three word elements, each defining a portion of ROM to be moved and finally terminated with three consecutive WRITEW(0) directives.

Now the project will compile and you will find in the .xMAP file the next information showing that the code and data were relocated in the intended location. Select **Project > Clean** option before compiling.

Listing 18. Code and Data Relocated at the Intended Location

```
# .my_ram
#>20000000          ___myRAMStart (linker command file)
  20000000 00000008 .myCodeInRAM CTMData(main.obj)
  20000008 00000015 .myCodeInRAM @18(main.obj)
  2000001D 0000001E .myCodeInRAM @19(main.obj)
  2000003B 00000030 .myCodeInRAM @20(main.obj)
  2000006C 00000000 .myCodeInRAM $t(main.obj)
  2000006C 00000040 .myCodeInRAM funcInROM(main.obj)
  2000009C 00000000 .myCodeInRAM $d(main.obj)
#>200000B6          ___myRAMEnd (linker command file)
```

5.2.4 Copying Code and Data from Flash to RAM

At this point you have relocated successfully the code and data in RAM. But the Microcontroller will not find them as they are still not copied into RAM. You can use the routine in the following listing to copy the code and data from Flash to RAM in runtime.

Listing 19. Copy Code and Data from Flash to RAM in Runtime

```
uint8 *Source; /* use this pointer to get the begining of the Code */
uint8 *Destiny; /* use this pointer to get the RAM destination address */
uint32 MemorySize; /* Gets the size of the code that is being copied */
void copyToMyRAM();//Function prototype
void copyToMyRAM(){
```

```
/* Initialize the pointers to start the copy from Flash to RAM */
Source = (unsigned char *)(CodeStart);
Destiny = (unsigned char *)(StartAddr);
MemorySize = (unsigned long)(CodeSize);
/* Copying the code from Flash to RAM */
while(MemorySize--)
{
*Destiny++ = *Source++;
}
}
```

Copy the code in the listing above in your main.c file before all the functions you already have. You must call `copyToMyRAM()` function inside `main()` function before you perform any operation.

NOTE This routine is needed only when a new memory segment is created as in the above case. If we only have a new section created which is in turn linked to .app_data then this routine is not required.

Copying of data from Flash to RAM can also be accomplished as shown in Listing 20, without using the user code in Listing 19. The three extra lines tells the startup routine to copy the data in my_ram.

Listing 20. Relocating .romp to copy data from Flash to RAM

```
.romp : AT(_romp_at)
{
__S_romp = _romp_at;
WRITEW(___ROM_AT);
WRITEW(ADDR(.app_data));
WRITEW(SIZEOF(.app_data));
WRITEW(___CodeStart);
WRITEW(___myRAMStart);
WRITEW(___CodeSize);
WRITEW(0);
WRITEW(0);
WRITEW(0);
}
```

6 Relocating Code and Data in External MRAM

Many times the internal RAM in the Microcontroller you are using is not enough for the application. For this reason it is needed to use external memories as part of the solution. The process to relocate code and data in external memories is exactly the same as you did for internal RAM. The only difference is that the external device needs to be communicated by an interface controller.

7 Creating a LCF for RAM Project

The difference in the ROM project and RAM project is that in this case, both code and data resides in the RAM and there need not be any copy of ROM to RAM. So .romp and .cfmprotect sections are not required in the LCF.

The following listing shows the memory distribution.

Listing 21. Memory segment

```
 MEMORY
{
m_interrupts  (RX) : ORIGIN = 0x1FFF0000, LENGTH = 0x000001E0
m_text        (RX) : ORIGIN = 0x1FFF01E0, LENGTH = 0x0001FE20
m_data        (RW) : ORIGIN = AFTER(m_text), LENGTH = 0x00020000
  }
```

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

Document Number: AN4498

30 January 2013